Distributing Application and OS Functionality to Improve Application Performance

Arthur B. Maccabe, William Lawry, Christopher Wilson,* Rolf Riesen†

April 2002

Abstract

In this paper we demonstrate that the placement of functionality can have a significant impact on the performance of applications. OS bypass distributes OS policies to the network interface and protocol processing to the application to enhance application performance. We take this notion one step further and consider the distribution of application functionality to the network interface and/or the operating system. We illustrate the advantages of this approach by considering double buffering at the application level, a standard technique used to hide communication latency.

1 Introduction

Over the past several years, a number of protocols and/or application programming interfaces (APIs) have been proposed for high performance interconnects. In this paper we examine these proposals in the light of application performance. In particular, we consider the performance of applications that use the Message Passing Interface (MPI) [7] and how this performance can be improved by careful placement of functionality that is traditionally considered to be part of the operating system or application.

*A. B. Maccabe, W. Lawry, and C. Wilson are with the Computer Science Department, University of New Mexico, Albuquerque, NM 87131-1386. This work was supported in part by Sandia National Laboratories under contract number AP-1739.

†R. Riesen is with the Scalable Computing Systems Department, Sandia National Laboratories, Org. 9223, MS 1110, Albuquerque, NM 87185-1110

We start by considering a traditional, layered implementation of MPI and discuss the inefficiencies inherent in this implementation strategy. We then consider the advantages offered by OS bypass implementations like MPICH/GM. In reality, OS bypass involves migrating parts of the OS functionality to a programmable network interface card (NIC) and the application. Using double buffering as a simple example, we then demonstrate that applications can gain even better performance by also migrating part of the application functionality to the network interface.

The goal of this paper is to provide a concrete illustration of the benefits associated with migrating functionality between the OS, the application, and the NIC.

2 A Typical, Layered Implementation of MPI

Figure 1 illustrates a typical implementation of the MPI communication stack. The application makes calls to the MPI library which makes calls to the socket library. The socket library, in turn, makes calls to the operating system using the trap mechanism. The operating system uses an internal protocol stack (TCP/IP) to handle the communication. Eventually, this involves a network device driver which uses a NIC to interface with the physical network.

There are at least three significant performance issues associated with this approach: first, it presents a potential for a large number of memory copies; second, it adds latency to message transmissions; and third, it adds significant overhead to message passing. We discuss each of these issues in turn.

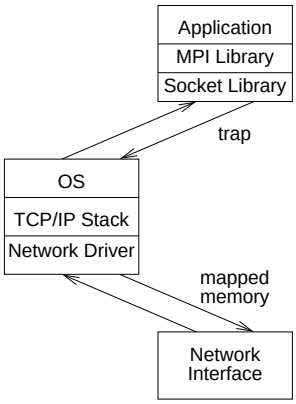

Figure 1: A Typical MPI Stack

2.1 Memory Copies

Memory copies have long been recognized as a significant problem in the implementation of communication protocols [11]. Memory copies lead to diminished communication bandwidth, increased communication latency and increased overhead associated with communication. Because of these costs, the communication stacks in most operating systems have been optimized to eliminate memory copies wherever possible.

Many memory copies result from the fact that message packets arrive with headers and data in a single contiguous region of memory. Early implementations of layered communication protocols used memory copies to separate the header and pass the data to the next higher level of the protocol. Current operating systems use buffer structures to avoid these memory copies. As an example, the Linux kernel uses the `skbuf` structure to avoid copies between protocol layers. Incoming packets are kept in the same kernel buffer while this data is being processed by kernel level protocols. When a lower level protocol passes a packet to a higher level protocol, it simply passes a pointer to the data portion of the current packet. This passing of pointers to avoid copies continues until the packets are reassembled into the application level buffer. This reassembly necessarily involves a copy into the buffer which was previously allocated by the application.

The construction of outgoing packets, in which each protocol layer needs to add a header to the data supplied by higher level protocols, can also result in additional memory copies. The Linux `skbuf` structures are also used to avoid these copies. Each protocol layer simply builds its header and prepends a pointer to this header to the list of buffers that is passed to the next lower layer. In most cases, a copy is only incurred as the final packet is prepared for transmission. If the network interface supports "gather" operations, even this copy can be avoided.

We should note that memory copies are most problematic when network transmission rates approach or even exceed memory copy rates. This situation first arose with the introduction of gigabit networking infrastructures. As memory copy rates have improved and are once again much higher than network transmission rates, it could be argued that elimination of memory copies is not as critical as it once was. While there is some truth to this argument, it is important to realize that memory copies contribute to communication latency and overhead.

2.2 Communication Latency

Latency is defined as the time needed to transmit a zero length message from application-space on one node to application-space on another node. At first glance, one might be concerned by the number of software layers involved in the processing of messages. However, these layers can be designed to reduce latency and, more importantly, will benefit from increases in processor clock rates. There are two more important sources of latency implicit in the strategy illustrated in Figure 1: traps and interrupts.

Because all message packets are passed through the operating system, communication latency will include the overhead of a trap into the operating system on the sending node to initiate the transmission and an interrupt on the receiving node to complete the transmission. The times for both of these operations (the trap and the interrupt) are fairly significant. Typical trap times for modern operating systems on RISC processors are one to two microseconds. Interrupt times, because interrupts need to traverse the I/O bus, tend to be significantly larger. Typical interrupt latencies for modern computing systems tend to be between five and ten microseconds. More important than the specific numbers, these delays will not improve at the same rate that processor speeds or network transmission rates will improve.

2.3 Communication Overhead

Communication overhead is the processing used to handle communication. This is processing capacity that is not available to the application. It has been noted that applications are more sensitive to increases in communication overhead than other parameters like transmission rate or latency [6].

Because the host processor is involved in every packet transmission, the strategy shown in Figure 1 implies a significant amount of overhead. In particular, the host processor will receive a large number of interrupts to handle incoming packets. Through indirect measurement, we have observed that interrupt overhead for an empty interrupt handler is between five and ten microseconds on current generation computing systems. Here, we should note that interrupt latency and interrupt overhead are independent measures even though their current values may be roughly the same. Interrupt overhead measures the processing time taken from the application to handle an interrupt. Interrupt latency measures how quickly the computing system can respond to an interrupt. By throwing away all work in progress on a pipelined processor, a CPU designer could reduce the latency of an interrupt; however, the interrupt overhead would not be affected as the work that was discarded will need to be reproduced when the application is re-started after the interrupt has been processed.

If not carefully controlled, communication overhead can quickly dominate all other concerns. As an example we consider minimum inter-arrival times for 1500 byte Ethernet frames. Table 1 summarizes frame inter-arrival times for several network transmission rates. As an example, the inter-arrival time for 100Mb Ethernet was calculated as:

$$1500\text{B} \times \frac{8\text{b}}{\text{B}} \times \frac{1}{100\text{Mb/s}} = 120\mu\text{s}$$

Assuming that interrupt overheads are 10 microseconds, including packet processing, we see that the communication overhead will be approximately 83% (10 out of every 12 microseconds is dedicated to processing packets) for gigabit Ethernet!

Two approaches are commonly used to reduce the communication overhead for gigabit Ethernet: jumbo frames and interrupt coalescing. Jumbo frames increase the frame size from 1500 bytes to 9000 bytes. In the case of gigabit Ethernet, this increases the minimum frame inter-arrival time from 12 microseconds to 72 microseconds which will reduce the communication overhead substantially. Unfortunately, this only works for larger messages and will not be particularly helpful for 10 gigabit Ethernet.

Table 1: Minimum Inter-arrival Time for 1500 Byte Ethernet Frames

Network Rate	Inter-arrival Time
10 Mb/s	1200 μsec
100 Mb/s	120 μsec
1 Gb/s	12 μsec
10 Gb/s	1.2 μsec

Interrupt coalescing holds interrupts (at the NIC) until a specified number of packets have arrived or a specified period of time has elapsed, which ever comes first. This reduces communication overhead by throttling interrupts associated with communication. When a constant stream of frames is arriving at a node, interrupt coalescing amortizes the overhead associated with an interrupt over a collection of frames. This approach will work well, with higher speed networks, but introduces a high degree of variability in the latency for small messages.

3 OS Bypass, Really OS Offload

To avoid the problems associated with memory copies, communication latency, and communication overhead, several groups have proposed protocols that support "OS bypass" [3, 12, 4, 8]. Figure 2 illustrates the basic strategy of OS-bypass. In concept, each process gets its own virtual network interface which it can use directly (bypassing the need to go through the operating system). In this strategy, memory copies are avoided by using buffers in the application rather than buffers in the operating system (as such, the application is in control of when copies need to be made). Communication latency is reduced by avoiding the need to trap into OS for sends and interrupt the OS for receives. Communication overhead is also reduced by the elimination of interrupts and because any processing

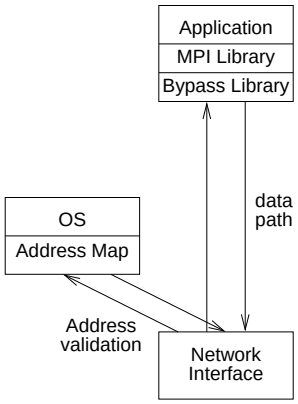

Figure 2: OS Bypass

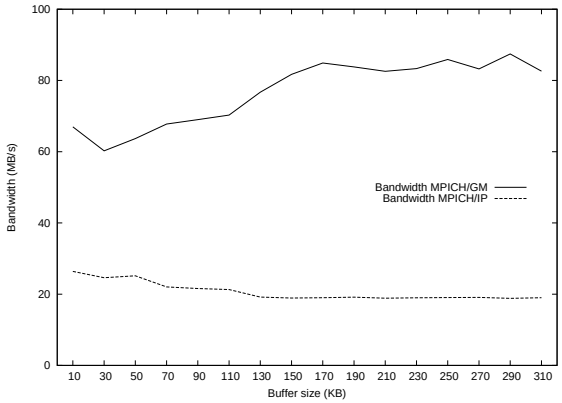

Figure 3: Bandwidth Improvement from OS Bypass

of packets from the network is handled by the application and, as such, should be directly related to the needs of the application.

Figure 3 illustrates the bandwidth improvement[1] due to OS bypass. This figure shows two bandwidth curves measured using a standard MPI ping-pong test. Both tests used the standard GM[2] (Glenn's Message) control program supplied by Myricom. Because this control program supports IP (Internet Protocol) traffic as well as GM traffic, we were able to run two different versions of MPI using the same underlying environment. The first, MPICH/IP, uses a traditional MPI implementation in which all communication is routed through the operating system. The second, MPICH/GM, uses OS bypass. The peak bandwidth improvement, from approximately 20 MB/s to approximately 80 MB/s, is significant.

As shown in Figure 2, OS bypass does not really bypass the OS. In particular, the NIC will need to rely on the OS for the translation between the logical addresses used by the application and the physical addresses used by the NIC. In addition, the NIC will most likely need to enforce other protection policies for the OS, e.g., limiting the nodes that the application can send messages to. For these reasons, we prefer to think of OS bypass as "OS Offload." That is, we are offloading part of the OS to the NIC. In particular, the part of the OS that is associated with communication policies. More than merely offloading access control related to communication to the NIC, OS bypass also offloads protocol processing to the application. In particular, application level libraries will be responsible for fragmenting and de-fragmenting messages to meet the requirements of the physical network. Figure 4 presents a graphical interpretation of this perspective.

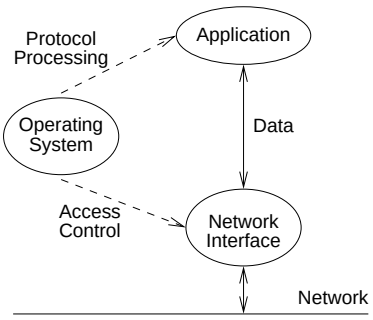

Figure 4: OS Offload

[1] All experimental results presented in this paper were obtained using two 500 MHz Pentium III processors running a Linux 2.2.14 kernel. These machines were connected through a single Myrinet switch and had LANai 7.2 NICs.

[2] All GM results are based on GM version 1.4. All MPICH/GM results use version 1.2..4

4 Application Offload

An important advantage of our perspective is that it allows us to consider other forms of offload. In particular, application offload[3], in which part of the application's activity is offloaded to the OS or the NIC. This observation led to the development of Portals 3.0, an API developed to support application offload [1].

The earliest version of Portals was implemented in SUNMOS [5] (Sandia/UNM Operating System), a lightweight, message passing kernel developed jointly by researchers at Sandia National Laboratories and the University of New Mexico. A later version of Portals was implemented in Puma [10], a follow-on to the SUNMOS kernel. Each of these early implementations relied on easy access to application memory while handling network events. In particular, these versions of Portals assume that the network interface is managed by a processor with direct access to application memory. While this was true for the Intel Paragon and Intel Teraflop [9] machines, it is not true in the commodity or near commodity networking technologies such as Myrinet, Gigabit Ethernet, and Quadrics that were considered for the Cplant [2] system. In these technologies, the network interface is managed by a processor on the PCI bus which has limited access to application memory.

An important goal in developing the Portals 3.0 API was support for efficient implementations of the upper level protocols that would be built using this API. Because MPI has become ubiquitous in high-performance computing, this meant supporting the protocols commonly used in MPI implementations along with the collection of protocols used to manage applications in a large scale computing system (e.g., application launch).

The "long message protocol" is perhaps the most critical protocol used in many implementations of MPI. In this protocol, the sending node generates a "request to send" (RTS) message which includes all of the MPI matching criteria and sends this message to the receiving node. The receiving node examines the contents of the RTS message and, when a matching receive has been posted, sends a "clear to send" (CTS) message. Once the CTS has been delivered, the sending node can transmit data packets as needed to complete the message transfer. Figure 5 presents a timing diagram for the long message protocol.

In examining Figure 5, notice that the data transfer portion of the protocol is handled by OS bypass. However, the initial stages, require a great deal of intervention from the upper level protocol which is implemented in the application (in the MPI library). First, the incoming RTS needs to be matched with a posted MPI receive to generate the CTS message. Second, the CTS needs to be matched with an earlier RTS to initiate the transfer of data. OS bypass does not provide a mechanism for implementing either of these matches.

Just as we would like to bypass the operating system and have the NIC control the flow of data between the network and the application, we would also like to have the NIC handle the matching used in the long message protocol. When the processor that controls the network interface has direct access to application memory, matching the RTS message to MPI receives posted by the application is relatively straightforward, the network processor can simply examine the data structures used to record posted receives. To match incoming CTS messages with previously sent RTS messages, the network processor needs to either cache information about RTS messages or access the memory where the application keeps this information.

This matching is far more complicated when the processor controlling access to the network interface is on the PCI bus. To avoid the need for direct access to the application's memory, the Portals API makes it possible to push the information needed to complete this matching onto the NIC. As such, the information needed to complete the matching is readily available when an RTS or CTS message arrives. This aspect of application offload is illustrated in Figure 6.

5 Double Buffering

To present a concrete example of the benefits associated with application offload, we consider double buffering in an MPI application. In this case, the application consists of two processes, a *producer* and a *consumer*. The producer supplies a stream of double precision floating point values and the consumer calculates a running sum of the values it receives. If the producer were to send each num-

[3]Because of it's relation to OS bypass, application offload has, on occasion, been referred to as application bypass.

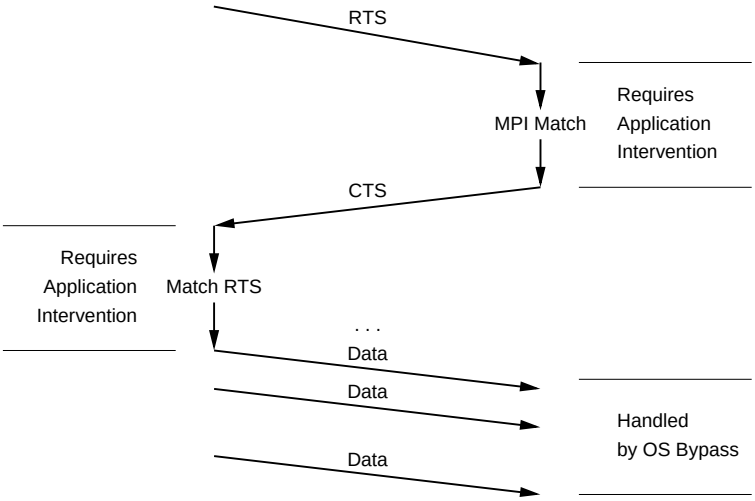

Figure 5: The Long Message Protocol of MPI

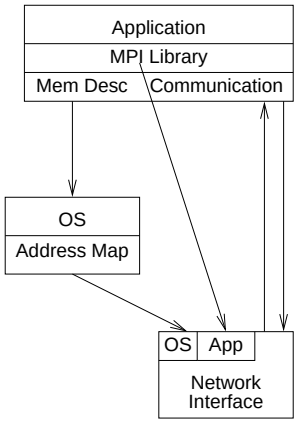

Figure 6: Illustrating OS and Aplication Offload

ber in a separate message, running time would be dominated by communication overhead. Instead, we have the producer prepare a batch of numbers which are then sent to the consumer. Moreover, the consumer will provide two buffers so that the producer can fill one buffer while the consumer is processing the other buffer. Pseudo-code for the producer and consumer processes is presented in Figure 7.

In examining the code presented in Figure 7, notice that the producer uses non-blocking sends to transmit filled buffers. This should allow the producer to overlap its filling of one buffer with the sending of the previously filled buffer. Similarly, the consumer uses pre-posted, non-blocking receives. When the producer is faster than the consumer, this should allow the consumer to overlap the processing of one buffer with the reception of the next buffer.

Figure 8 compares the performance of our double buffering application when it is run on MPI over Portals versus MPI over GM. While Portals supports application offloading, GM supports OS bypass. Each graph shows the rate at which the double buffering application is able to process data based on the size of the buffers. For both environments, the throughput peaks with buffers of about 10 kilobytes. Once this peak is attained, the

```
Producer:                          Consumer:
double A[BSIZE], B[BSIZE];         double A[BSIZE], B[BSIZE];

fill A;                            ireceive A; isend CTS A;
wait CTS A; isend A;

fill B;                            ireceive B; isend CTS B;
wait CTS B; isend B;

for( i = 0 ; i ¡ n-1 ; i++ ) {     for( i = 0 ; i ¡ n ; i++ ) {
   wait A sent; fill A;            wait A received; sum A;
   wait CTS A; isend A;            ireceive A; isend CTS A;

   wait B sent; fill B;            wait B received; sum B;
   wait CTS B; isend B;            ireceive B; isend CTS B;
}                                  }
```

Figure 7: Pseudocode for Double Buffering

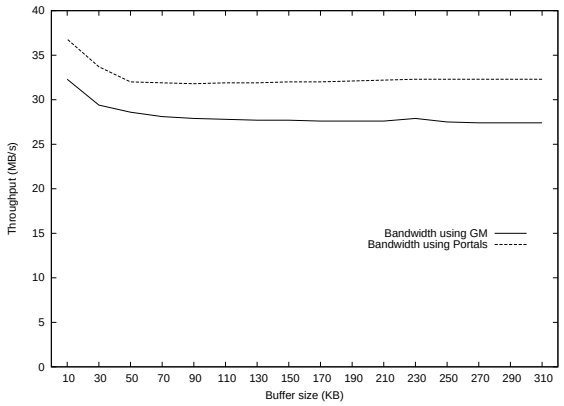

Figure 8: Double Buffering Performance (Producer Bounded)

throughput over Portals is approximately 15% better than the throughput attained over GM.

The intention of double buffering is to allow the producer and consumer to run in parallel. Because we are using non-blocking sends and pre-posted receives, the transmission of the data from the producer to the consumer should also occur in parallel. Ultimately, the overall rate will be limited by the slowest among the producer, the network, and the consumer. Figure 9 presents a Gantt chart to illustrate the three possible bottlenecks that might be encountered in double buffering.

In our experiment, we measured the producer rate at approximately 43 MB/sec and the consumer rate at approximately 123 MB/sec. Both measurements were taken independently of any communication. (The producer is significantly slower because it calls the random number generator for each double precision value used to fill the buffer.) The network rate for MPICH/GM is over 80 megabytes per second and that for portals is over 50 megabytes per second. As such, the producer represents the bottleneck.

In examining Figure 8, note that the peak processing rate for double buffering over Portals is approximately 38 MB/sec. The peak processing rate for double buffering over GM is approximately 32 MB/sec. While the Portals processing rate is close to the producer rate, the GM processing rate is significantly lower than the producer rate.

Before we consider the poor performance of double buffering over GM, we should note that the double buffering over Portals is actually a bit better than it appears. When we measured the producer and consumer rates, we measured them in absence of communication. Because

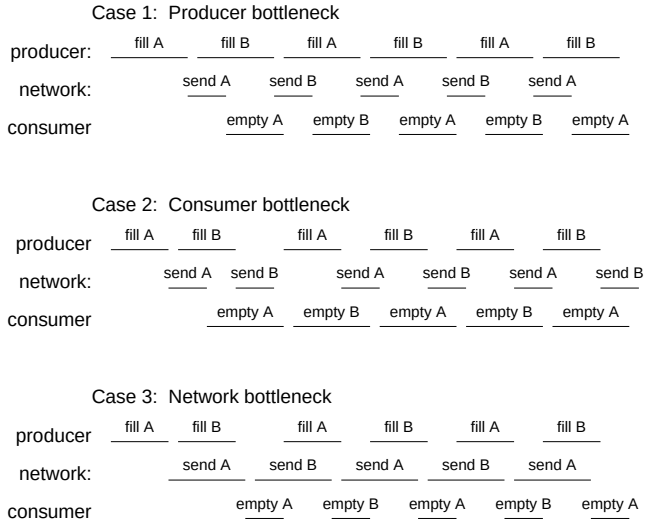

Figure 9: Overlap in Double Buffering

the current implementation of Portals does not support OS bypass, i.e., all communication goes through the host OS, the actual producer and consumer rates are lower than the numbers reported earlier. We will consider this more in more detail later in this paper.

5.1 Why Double Buffering over GM Performs Poorly

Figure 10 presents a Gantt chart to explain the relatively poor performance of double buffering over GM. When the producer finishes filling the first buffer, it initiates the transfer of this buffer to the consumer. This causes the MPI library to send an RTS message to the MPI library at the consumer. Because the consumer is waiting (in the MPI library) for its first buffer to be filled, the CTS is immediately returned to the producer. Unfortunately, by the time the CTS arrives at the producer, the producer has left the MPI library and is filling the next buffer. Thus, the first buffer is not sent until the second buffer is filled and the producer calls the MPI library to transmit the second buffer. Because the producer cannot re-fill the first buffer until it has been sent, the producer is blocked until the first buffer is sent. Immediately after sending the first buffer, the MPI library at the producer sends an RTS for the second buffer. Again, the CTS for this buffer does not arrive until the producer has begun filling the next buffer. While this pattern allows overlap in the execution of the producer and consumer, it does not result in overlap of the communication with either of these processes.

In particular, the implementation strategy of MPICH/GM requires that the application make a call to an MPI library routine to match the incoming CTS with the previously sent RTS. That is, the application intervention required on the left side of Figure 5 cannot be handled by the NIC and can only be attained by having the application make a call to the MPI library. In contrast, the MPI implementation over Portals takes advantage of application offloading to match the incoming CTS and initiate the data transfer *while* the producer is filling the next buffer.

5.2 Further Experimentation

In our initial experiment, the producer rate was much lower than the consumer rate (43 MB/sec versus 123 MB/sec). Noting that the overall production rate is bounded by the minimum of the producer rate and the consumer rate, we ran an experiment in which the producer rate was higher than the consumer rate. In order

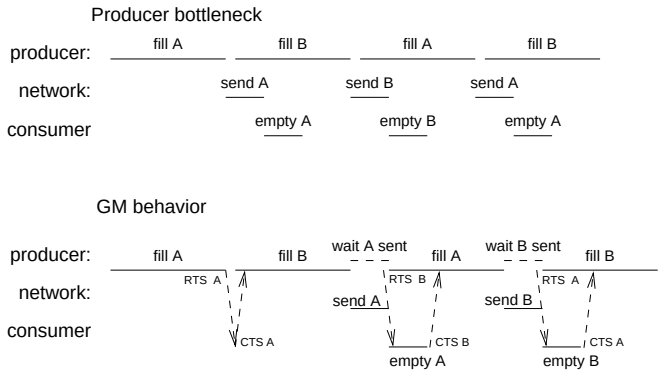

Figure 10: GM Behavior

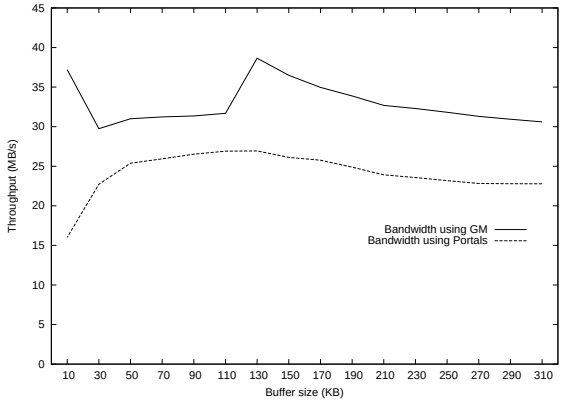

Figure 11: Double Buffering Performance (Consumer Bounded)

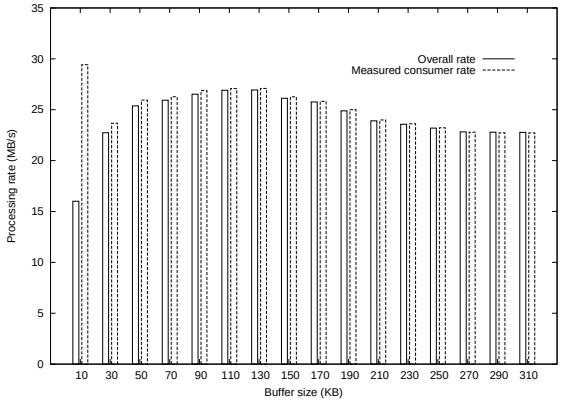

Figure 12: Measured Consumer Rates for Portals

to control the producer and consumer rates, we replaced the code to fill buffers (in the producer) and empty buffers (in the consumer) with a variable length delay loop. Figure 11 presents the results of an experiment in which the producer rate is approximately 160 MB/sec and the consumer rate is approximately 40 MB/sec.

In this case, we note that the overall processing rate for double buffering over GM is significantly better than the processing rate for double buffering over Portals. This is due to the fact that the current implementation of Portals does not use OS bypass. That is, all packets and data are transmitted through the OS and the OS makes copies of each data packet. This has a significant impact on the actual processing rates. To examine this aspect of GM and Portals, we instrumented the test code to measure the actual consumer rates. Figures 12 and 13 compare the overall processing rate with the measured consumer rates for double buffering over Portals and GM, respectively.

In comparing these graphs, notice that the overall processing rate for double buffering over Portals is very close to the measured consumer rates. That is, the implementation over Portals is very close to optimal. In contrast, the relationship between the overall processing and consumer

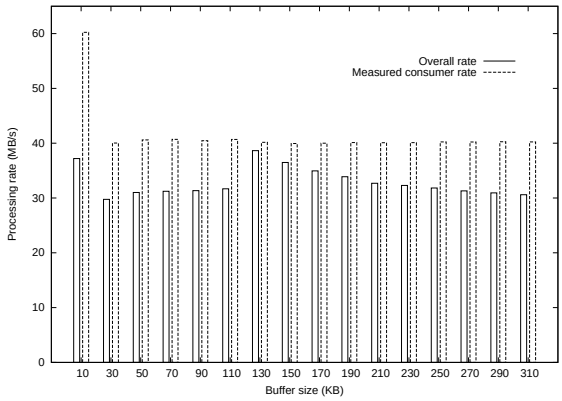

Figure 13: Measured Consumer Rates for GM

rates for double buffering over GM does not exhibit this same behavior.

6 Discussion

It is our contention that the results presented in this paper represent strong evidence supporting the importance of application offloading. That is, moving part of the functionality that is associated with an application into the operating system and/or special purpose processor (in this case a communication processor). There are three issues that should be addressed when considering these results. First, we will consider the significance of application offloading. Second, we will consider the significance of our experimental application, double buffering. Finally, we will consider other approaches to providing application offloading.

In a sense, the application offloading described in this paper essentially provides good support for an upper level RTS/CTS flow control protocol. More specifically, the RTS/CTS protocol used to transmit long messages in MPI. Given the specific nature of our presentation, it makes sense to ask how significant is application offloading. Our answer to this question comes in two parts. First, MPI is a widely used message passing API and the long message protocol is one of two basic protocols used in the implementation of MPI. Second, RTS/CTS flow control protocols are used in a variety of upper level protocols (e.g., file servers) and the application offloading supported by Portals can be used in the implementation of these protocols as well.

All of our experimental results are based on the performance of double buffering. This is because double buffering is a well known technique that permits the maximal amount of parallelism at the algorithmic level. It is up to the underlying system to take advantage of the parallelism expressed by the application. In this respect, double buffering is a fair test of the underlying runtime system. We have shown that an implementation of MPI over Portals is able to exploit all of the parallelism expressed by the application while MPICH/GM is only able to exploit a fraction of this parallelism. Another question is whether any applications actually takes advantage of the parallelism it could express. That is, do applications use double buffering? While this is a less significant question, it is somewhat valid. Double buffering is frequently used to hide the latency of network communication. In the past ten or more years we have seen that network latency is not scaling with improvements in network bandwidth or processor cycle times. This trend will increase the importance of latency hiding in applications. As such, whether or not it is explicit in applications, latency hiding will be used in many of the runtime environments (or frameworks) used by applications.

In this paper, we have emphasized the fact that the Portals interface was designed to support application offloading and it is reasonable to ask whether application offloading can be supported at other levels. The answer is certainly yes; however, other approaches carry a cost. The essential problem that application offload addresses is the fact that the application may be busy performing a calculation when it needs to participate in a communication protocol. In operating systems, the traditional way to deal with this kind of problem is to use interrupts. This strategy maps to “upcalls” or signals at the application level, i.e., part of the application is offloaded to a signal handler. Alternatively, we could consider adding a special purpose processor (as is done with intelligent NICs) to poll for activity. At the application level, this strategy would map to a thread for handling communication protocols, i.e., part of the application is offloaded to a special thread. While either of these approaches would be relatively easy to implement, we note that the MPI implementation on GM does not provide either. An important problem with each

of these approaches is the overhead incurred.

7 Future Work

Noting that the current implementation of the Portals API does not use OS bypass, there are several efforts underway to bypass the OS in future implementations of the Portals API. One group at Sandia National Laboratories is in the process of implementing the entire Portals API on Myrinet NICs. Another group at Sandia is examining the incremental migration functionality to Myrinet NICs and Quadrics NICs. A group at UNM is investigating the incremental migration of of functionality onto Gigabit Ethernet NICs.

8 Acknowledgements

Ron Brightwell from Sandia National Laboratories was instrumental in the development of the Portals 3.0 API and the key concepts associated with application offload. Tramm Hudson developed the reference implementation of the Portals API which enables easy migration of functionality between the application, OS, and NIC.

We would also like to thank the other members of the Scalable Systems Laboratory at UNM: Sean Brennan, Patricia Gilfeather, Dennis Lucero, Carl Sylvia, and Wenbin Zhu. Patricia and Wenbin in particular have been working on offloading related to TCP/IP and Portals, respectively. Their experiences have been critical in guiding the work reported in this paper.

References

[1] Ron Brightwell, Tramm Hudson, Rolf Riesen, and Arthur B. Maccabe. The portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.

[2] Ron B. Brightwell, , Lee Ann Fisk, David S. Greenberg, Tramm B. Hudson, Michael J. Levenhagen, , Arthur B. Maccabe, and Rolf Riesen. Massively parallel computing using commodity components. *Parallel Computing*, 26:243–266, February 2000.

[3] Compaq, Microsoft, and Intel. Virtual interface architecture specification version 1.0. Technical report, Compaq, Microsoft, and Intel, December 1997.

[4] Mario Lauria, Scott Pakin, and Andrew Chien. Efficient layering for high speed communication: Fast messages 2.x. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, 1998.

[5] Arthur B. Maccabe, Kevin S. McCurley, Rolf Riesen, and Stephen R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pages 245–251, June 1994.

[6] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 85–97, New YOrk, June 2–4 1997. ACM Press.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[8] Myricom, Inc. The GM message passing system. Technical report, Myricom, Inc., 1997.

[9] Sandia National Laboratories. *ASCI Red*, 1996. http://www.sandia.gov/ASCI/TFLOP.

[10] Lance Shuler, Chu Jong, Rolf Riesen, David van Dresser, Arthur B. Maccabe, Lee Ann Fisk, and T. Mack Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

[11] Peter A. Steenkiste. A systematic approach to host interface design for high-speed networks. *IEEE Computer*, 27(3):47–57, 1994.

[12] Task Group of Technical Committee T11. Information technology - scheduled transfer protocol - working draft 2.0. Technical report, Accredited Standards Committee NCITS, July 1998.